AFRL-IF-RS-TR-2005-153
Final Technical Report
April 2005

SCALING PROOF-CARRYING CODE TO PRODUCTION COMPILERS AND SECURITY POLICIES

Princeton University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. H559

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-153 has been reviewed and is approved for publication

APPROVED: /s/

JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2005	Final Jun 99 – Jun 04

4. TITLE AND SUBTITLE
SCALING PROOF-CARRYING CODE TO PRODUCTION COMPILERS AND SECURITY POLICIES

5. FUNDING NUMBERS
C - F30602-99-1-0519
PE - 62301E/63760E
PR - H559
TA - 10
WU - 01

6. AUTHOR(S)
Andrew W. Appel, Edward W. Felton,
David P. Walker, Zhong Shao, and
Valery Trifonov

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Princeton University
Department of Computer Science
35 Olden Street
Princeton New Jersey 08544-2087

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-153

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: John C. Faust/IFGB/(315) 330-4544/ John.Faust@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
We have developed high-assurance software protection mechanisms that can be used in component-software platforms (virtual machines such as Sun's Java or Microsoft's .Net). The idea is to leverage type-safe source languages to get fine-grained protection at the level of individual object fields and methods. The obstacle to be overcome was the inherent complexity of virtual machine implementations, particularly their just-in-time compiler, before installing that output for execution. We have successfully shown how to integrate this technology into the compiler, and how to conduct formal, automated verification of the protection mechanism.
We have also developed a technology for distributed authentication and public key distribution, called "proof-carrying authentication." This allows many participants with different goals, different policies, and even different policy languages, to cooperate in authenticating each other (and third parties).
Finally, we have started a seedling effort in policy-based network management.

14. SUBJECT TERMS
Proof-Carrying-Code, Software Protection, Virtual Machines, Java, Formal Verification, Distributed Authentication, Policy-Based Network Management, Language-Based Security

15. NUMBER OF PAGES
18

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1. Introduction

DARPA has supported research in formal verification for many years (see MacKenzie's book for an interesting history of how ARPA helped get this field started in the 1960s and 70s [MacKenzie 2001]). But formal verification has sometimes been criticized for a long list of shortcomings: it's too expensive to do; people don't verify the actual engineered systems but just simplistic abstractions of the real world; formal proofs of real systems are too big to check for errors; and so on. These criticisms cannot be lightly dismissed. What we set out to do in our DARPA-supported research was to apply formal verification in settings that can address these criticisms directly.

We chose two main problem domains: Language-based security for virtual machines, and distributed authentication frameworks. For each problem domain, I'll explain what is the problem that motivates the need for formal verification, why it seemed that formal methods might actually be useful in this domain, and what solutions we engineered.

2. Language-based security for virtual machines

Component-software platforms such as Sun's Java and Microsoft's .Net are increasingly used by commercial software developers because they provide many advantages over conventional development (in languages such as C and C++). Commonly accepted software-engineering practices such as data encapsulation (information hiding) are enforceable on these platforms, in contrast to C/C++ where careless programmers can bypass information-hiding rules. This brings many benefits: software development is quicker and software is more reliable. There are important security benefits as well: Java programs[1] are basically immune to buffer-overrun vulnerabilities, which is the most common path that attackers use to subvert software on the Internet today.

Because modern programming languages such as Java provide better support for reusable components than previous-generation languages, commercial developers often build their software from a combination of off-the-shelf and custom-built components. Often the off-the-shelf components are obtained from other parties. This leads to many efficiencies in building lower-cost and more reliable software; but it can lead to security worries. If you install software on your mission-critical system, you want to be sure that it does not have Trojan horses that will steal or corrupt your data. The people who write the software pose an "insider threat," but with contemporary component-software integration practices, there's a very wide set of "insiders" to consider.

[1] Almost everything I will say about Java applies as well to Microsoft's C# (pronounced C-sharp) language. The Java language is used to write programs that run on the Java Virtual Machine (JVM); the C# language is used to write programs that run on the Common Language Runtime (CLR) of Microsoft's .Net platform. JVM and CLR can run on conventional operating systems such as Linux and Windows; JVM can also run directly on embedded devices such as smart cards and cell phones.

A conventional (and very reasonable) solution to this problem is to use protection mechanisms so that less-trusted software components have limited access to sensitive data. In the past, when each component was an entire software application that ran in its own address space, we could use virtual memory, jointly implemented by the hardware and the operating system, as such a protection mechanism.

But virtual memory will not work well as a protection mechanism for Java components, because the more-trusted and less-trusted components all share the same address space and the same operating-system process. There are some very good reasons that components share an address space: it allows the interaction between components to use expressive and efficient object-oriented interfaces, instead of clumsy and slow remote procedure calls and message passing. Therefore virtual memory cannot serve as the protection mechanism.

Java has a built-in protection mechanism: its type-checker. Java's type system enforces, in the source code, fine grain access control by software components. Objects of one class cannot read or write the private fields of objects from other classes. Java source code is compiled to byte-codes (called "Java Virtual Machine Language", or JVML, but I will just use "byte-codes"), and it is the byte-codes that are shipped and installed on users' JVMs, but the byte-codes are also type-checkable by the byte-code verifier built into the JVM. In principle, the language-based protection built into Java can allow software to be built from a combination of less-trusted and more-trusted components, so that less-trusted components have limited access to data owned by more-trusted components.

But there's a big problem with language-based protection in conventional virtual machines for Java and C#. What if the protection mechanism has holes in it? That is, what if bugs in the implementation of the virtual machine allow the less-trusted programmers of less-trusted components to bypass the protection and access private fields of more-trusted components? To assess this problem, we examine what parts of a Java Virtual Machine are in the "trusted base" of the protection mechanism. The trusted base of a system is the part in which bugs could cause security vulnerabilities. We find [31] that the trusted base of a conventional JVM is huge, comprising hundreds of thousands of lines of code. One of the biggest problems is that the JVM contains a just-in-time (JIT) compiler that translates byte-codes to native machine code (e.g., Intel Pentium machine language); the JIT compiler is well over a hundred thousand of lines long in a high-tech JVM, and, as I will describe, a bug in the JIT could be exploited to breach the protection mechanism. It's not realistically possible to write 100,000 lines of code with no bugs at all, so we cannot rely on this protection mechanism.

A correct JIT compiler guarantees that well-typed byte code will compile to machine code that respects its interfaces; that is, the machine code output from the JIT will not access private fields of other objects. Bugs in the JIT compiler can be exploited by software components that run on the JVM platform, as follows. Consider an attacker who is providing one of the less-trusted components of a large component-software system written in Java. Normally, the program he writes will never be given access to

private data. However, suppose the unscrupulous programmer learns of a bug in the JIT compiler that causes well-typed byte-code to be (incorrectly) compiled to machine code that does not respect its interfaces. He writes the Java source code for his component to trigger this bug. The component he supplies is now able to bypass the protection mechanism.

This is a pity, because Java's object-oriented interfaces and language-based security lead to significant productivity and reliability improvements in software development, and if only we could rely on its language-based protection mechanism, we could build more-secure systems from reusable software components.

2.1 Research goal and approach

Thus we arrive at one of the goals of our DARPA-funded research project: *design and prototype an architecture in which the language-based protection mechanism is provided by a JIT compiler, in such a way that the protection mechanism can be formally verified.* The formal verification itself should be believable, that is, it should be mechanically checkable using a substantially smaller trusted base than the software being verified; and the verification should be of the actual software that is installed in some prototype system.

We had several reasons to believe that this problem could be successfully approached using formal verification, *using the right approach*. A brute-force approach would be to formally verify all the Java code that is sent to the JVM; but this would be impossible, because there's so much of it, and it's written by programmers who don't have the time or the expertise to do the verification. A slightly less impossible approach would be to verify the correctness of the JIT compiler; then just one formal verification (of the compiler) could be leveraged to guarantee safety properties of all the unverified, untrusted components that are compiled through it. But still, the compiler is too large for current formal-methods techniques to work well.

The approach we take is therefore *proof-carrying code:* we have the JIT compiler emit, along with the native machine code, evidence that the code is safe to execute (i.e., that it respects the security policy). The evidence is independently checked (and compared with the code). If the evidence doesn't check out, or fails to correspond to the code, then the code is rejected and is not installed.

This crucially relies on a *sound* logical system for checking the evidence. Soundness means, "if the checker accepts the evidence for the claim, then the claim is true." An important part of our research has been to demonstrate the soundness of our system.

This setup can safely tolerate bugs in the JIT compiler. A bug may cause the code to be unsafe, in which case it will be impossible to produce evidence that the checker will accept; or a bug may cause the evidence-generator to provide bad evidence, in which case the code will also be rejected. In the worst case, a bug may cause the compiler to

produce incorrect but safe output (machine code that doesn't work right, but at least obeys the security policy). In this case, it is possible that the evidence would check and that the code could be installed and executed.

2.2 Architecture of our design

The major components of our solution are as follows. We start with a program in a type-safe source language; the languages we have prototyped include ML and Java.

- Framework, type systems, and algorithms for propagating type-checking information from the front end of the compiler (Java or ML source code) to the middle of the compiler (intermediate language(s)). We have several major results in this area, covered in technical detail by several of our publications. [1,8,9,10,11,19,21,23,24,28,29,44,52,53]
- Framework, type systems, and algorithms for propagating type-checking information from the middle of the compiler to the back end of the compiler (where the machine code is produced); we have several major results on this topic [16,26,27,34,38,40,46,55]
- Design of a language for communicating "evidence" from the compiler to the independent checker. [46,54,55,38]
- Soundness proof: A formal proof that if the "evidence" is accepted by the checker, then the corresponding machine code must obey the safety property. This is one of the major novel results of our research; no previous proof-carrying code system had such a proof. Because the proof is so large, it must be checked by machine. Thus, this part of the research involves both the underlying mathematics of why soundness holds, and also the techniques for representing this mathematics in a machine-checkable system. [3,4,5,7,12,14,17,18,20,36,45,50,56,58]
- Design and implementation of the independent checker itself. This checker checks two things: (1) the soundness proof for the evidence language, and (2) the evidence for safety of a particular program. This checker is the "trusted base" of the entire system; if (and only if) it is correct, then the whole system can guarantee enforcement of the protection mechanism. Therefore it's very important that the checker itself be small and simple enough that it can be implemented correctly. Our checker is small and simple: it's about 1,100 lines of C code. [30,54]

We can make quantitative measurements of various components. The SML/NJ compiler from ML source code to Sparc machine language is approximately 90 kloc (thousand lines of code). Thus, for a conventional compiler the implementation effort (where smaller is better) was 90 kloc, and the trusted base (where smaller is better) was also 90 kloc.

Our new proof-carrying-code version of the SML/NJ compiler is approximately 114 kloc; that is; approximately 24 kloc is devoted to producing the "evidence" for the checker. The specification of the "evidence language" is 5 kloc, and the soundness proof is about 134 kloc. Thus, there is a total of 253 kloc of untrusted code; that is, a bug in any part of that code would not lead to a vulnerability, because if it caused unsafety it would be caught by the checker. We have a trusted base of 3 kloc: the specification of the safety policy and Sparc machine architecture is 1.8 kloc, and the checker is 1.1 kloc.

Overall, the project has been successful, not only at demonstrating the applicability of mechanized formal verification to the assurance of protection mechanisms in virtual machines, but also to the measurement of how large a task it is. What we find is that, in this case, formal verification reduces the trusted base by a factor of approximately 30, and increases the development effort by a factor of approximately 3.

Increasing the development effort by a factor of 3 is not a tradeoff to make lightly. We argue that for a highly leveraged piece of software (such as the virtual machine that is used as platform for many application modules) the investment may be worthwhile.

2.3 Other investigations in language-based security

In addition to the Foundational Proof-Carrying Code project described in the previous subsection, we also investigated how a wider class of security policies could be expressed using techniques that go beyond ordinary type checking. We have found a variety of static techniques (that can analyze programs before they execute) [12, 39, 48, 59] and dynamic techniques (that monitor programs as they execute) [32, 49].

We also performed an experiment to question the very assumptions on which proof-carrying code (and language-based security) is based. Our soundness proof relies upon the assumption that the computer actually executes its specified instruction set. What if the attacker tried to exploit the fact that sometimes this is not the case? In particular, it is known that machines sometimes have transient memory errors: a bit in memory will change its value, perhaps because a cosmic ray passes through the memory chip. This invalidates our assumption; the question is, could an attacker actually take advantage of this to breach confidentiality and integrity of protected data? We show that the answer is yes [35]; one can design a Java program that takes advantage of random memory errors to defeat the language-based protection. However, we also show that conventional hardware techniques (parity, or single-error-correction-double-error-detection) are adequate to restore the protection guarantee.

We have continued investigations into whether hardware errors could compromise in other ways the assumptions under which language-based security operates. For example, we have tried attacks suggested by recent research on the susceptibility of processors to software-induced voltage swings [Joseph 2003]. We have found that modern commercial microprocessors are robust against such attacks. Overall, our conclusion is that language-based security is robust with respect to the assumption that the computer hardware executes its instruction set as specified, but that this issue should not be ignored.

3. Proof-carrying authentication and access control

Access control is a nontrivial problem even in a single-host system. One must authenticate the users (that is, determine whether they are who they say they are) and then determine whether those users have permission to read (or write, etc.) the given data. But in a distributed setting, the problem is much more complicated. There are many different owners of data, each of which must make access-control decisions for many different (overlapping) groups of users. Different data-owners may have different policies; in fact, they may have different languages for expressing policies. Some parties to the system play specialized roles such as certifying the association of authentication keys to users.

Previous work on distributed authentication logics [Lampson 1992] and on public-key infrastructures [Housley 2002, Ellison 1999] has given us a useful framework for thinking about the problem, but each of these works assumes that there is just one language for expressing policies—and, of course, each work has its own different idea of what that language should be.

3.1 A distributed authentication framework

The problem we set out to solve is this: build a framework for specifying and implementing languages for distributed authentication/authorization frameworks, so that different languages (even ones yet to be invented) can interoperate soundly, and so that expressive languages can be designed for application-specific purposes.

The approach we take is this: for each policy language, define its operators using formal logic. (The user of the language won't have to see the formal logic.) Then use these definitions to prove the soundness of each policy language individually. Because each language is definitionally specified in the same underlying logic, they can then soundly interoperate, not only with each other, but with policy languages yet to be invented. [2, 13]

Our system relies on machine-checked formal proof, just as our proof-carrying code system does. We use the same logical infrastructure as in PCC, and the same proof-development tools. In fact, the same (small and simple) independent checker that we used for PCC can also be used for proof-carrying authentication.

We built the framework, and to demonstrate its effectiveness we used it to implement a distributed access-control policy language. The prototype language has features such as certificates, local names, general delegation, specific delegation, and time-dependent policies. We show how it can be easily extended to do expiration, key management and revocation, abstract principals, and delegation across domains.

We interfaced our implementation to the Apache web server and to a proxy web client, and we showed that it is useful in practice in fine-grain access control to web pages on the Internet. [60, 37]

3.2 Proof-carrying linking

Another application of proof-carrying authentication is the specification and checking of software component assemblies.

Large software systems are often built from loosely coupled subsystems. When a software integrator uses a third-party software component as a building block in a system, he doesn't want the code he imports to break the whole system. How can he determine that it is safe to link the foreign software component into the system?

The most widely used methods for ensuring safe linking are type checking and code signing. Type-checking can make some very strong guarantees of some security policies—it is the basis of the our PCC research, for example; but we also wanted to explore how code signing could be used in a principled way. Code signing ensures that someone trustworthy trusts the code, but it is not always enough for guaranteeing system safety since trusted software companies or software developers unintentionally make mistakes. Software signed by trusted companies can still cause security holes in users' systems. For example, in November 2002 Microsoft released a badly coded ActiveX control, signed (as usual) with Microsoft's code-signing key; the bug led to a security vulnerability. Because Microsoft's code-signing protocol is insufficiently expressive, Microsoft was faced with the choice of setting a *kill bit* so that no browser would run the control—thereby disabling thousands of websites, even ones containing no security-critical data---or not setting the bit—thereby continuing to *endorse* the product. Microsoft chose the latter; it recommended that users who desired a secure system should remove *Microsoft* from Internet Explorer's Trusted Publisher List [Microsoft 2002].

We designed and prototyped Secure Linking (SL) [33, 42], a flexible way of allowing software component users to specify their security policy at link time, giving the users more control than type-checking or traditional digital signing. Our Secure Linking mechanism would not prevent bugs in ActiveX in the previous example, but it would give the software provider and the software consumer finer-grained control of the meaning of certificates they use.

With the SL framework, a code consumer can establish a linking policy to protect itself from malicious code from outside. The policy can include certain properties that the code consumer thinks useful for system safety: software component names, application-specific correctness properties, version information of software components, and so on. To link and to execute a component in a SL-enabled system there must be a machine-checkable proof that the component has the properties specified in the code consumer's linking policy. This proof might be provided by the code provider, or might be produced by an untrusted proving algorithm that runs on the code consumer's machine. The proof is formed using the logic and inference rules of the framework. After being submitted, the proof is checked by a small trusted proof checker in the code consumer, and if verified, the component is allowed to be linked to other components in the code consumer.

Just as our proof-carrying authentication/authorization framework is designed to accommodate many different policy languages, so is our SL framework. We tested this aspect of SL by showing that Microsoft's "package assembly" language, for specifying linking policies in their .Net framework, can be encoded in SL and interoperate with other policy languages.

4. Information assurance measurement (policy-based network management)

In early 2003, we began a seedling project to investigate whether mechanized logical reasoning could be helpful in dealing with the complexities of network management, with firewalls, routers, server software, CERT advisories, and so on. We have made substantial progress on this effort, leading to a significant research result in November 2004, a few months after the end of the DARPA contract [61].

To determine the security impact that software vulnerabilities have on a particular network, one must consider interactions among multiple vulnerabilities and multiple hosts. We implanted a tool called MulVAL, an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network. MulVAL is cleanly modularized so that it can effectively leverage existing work on single-host vulnerability recognition and network configuration management. The reasoning engine in MulVAL takes inputs from off-the-shelf tools [Wojcik 2003] and performs analysis on the whole network to determine if vulnerabilities found on individual hosts can result in a condition violating a given high-level security policy. MulVAL considers interaction among multiple hosts and multiple vulnerabilities, as well as network and machine configurations that are relevant in attacks. In the wake of a new vulnerability report, MulVAL can quickly tell a system administrator if the new bug breaks the security policy of the network. If so, MulVAL generates an attack trace to help the system administrator decide upon countermeasures.

5. Technology transfer

Our research can be most useful to DARPA and its clients if our technology can be transitioned into industrially useful tools. We have actively sought avenues for technology transfer.

Our proof-carrying code research shows how to build more-secure, higher-assurance virtual machine platforms for languages such as Java and C#. The natural technology transfer path is to have the commercial builders of such platforms learn how to adopt this technology. Then, as a matter of course, any existing or new Java or C# applications can and will be run on more-secure platforms. For the Java platform, we have collaborated since 2001 with Intel Research, who make open-source Java and .Net research virtual machines. They have been incorporating some of the evidence-generation technology in

their JIT compiler. We have also worked informally with researchers at Microsoft who are constructing their next-generation virtual machine for the .Net platform. In fact, the graduate student who built our certifying compiler back end [55] is now working on similar projects at Microsoft research.

The technology transfer for proof-carrying authorization is being done in a different way. The student who built our PCA system [37] is now a research scientist working on a project led by Michael Reiter at CMU to build PCA into smart cards for a distributed access-control system. This project should demonstrate whether PCA can be successful in demanding circumstances.

We are planning to do technology transfer of policy-based network management based on a collaboration with HP Laboratories. We have been discussing this with them for several months and serious work on a joint project is likely to begin in early 2005.

Bibliography.

Numbered items are publications resulting from this research contract. Items in author-year format [MacKenzie 2001] are works by other researchers cited in this report.

[1] Representing Java Classes in a Typed Intermediate Language, by Christopher League, Zhong Shao, and Valery Trifonov, ACM SIGPLAN International Conference on Functional Programming (ICFP'99), September 1999.

[2] Proof-Carrying Authentication, by Andrew W. Appel and Edward W. Felten, 6th ACM Conference on Computer and Communications Security, November 1999.

[3] A Semantic Model of Types and Machine Instructions for Proof-Carrying Code, by Andrew W. Appel and Amy P. Felty. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00), January 2000.

[4] Algorithm-Independent Framework for Verifying Integer Constraints. David Teller and Zhong Shao. Technical Report YALEU/DCS/TR-1195, Department of Computer Science, Yale University, March 2000.

[5] Machine Instruction Syntax and Semantics in Higher Order Logic, by Neophytos G. Michael and Andrew W. Appel, in CADE-17 (Conf. on Automated Deduction), June 2000.

[6] Type-Preserving Garbage Collectors, by Daniel C. Wang and Andrew W. Appel. 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01), January 2001.

[7] Efficient Substitution in Hoare Logic Expressions, by Andrew W. Appel, Kedar N. Swadi, and Roberto Virga. 4th International Workshop on Higher- Order Operational Techniques in Semantics (HOOTS 2000), September 2000.

[8] Fully Reflexive Intensional Type Analysis, by Valery Trifonov, Bratin Saha, and Zhong Shao, in 2000 ACM International Conference on Functional Programming (ICFP'00), Montreal, Canada.

[9] Type-Preserving Compilation of Featherweight Java, by Christopher League, Valery Trifonov, and Zhong Shao. Foundations of Object Oriented Languages (FOOL '01), January 2001.

[10] Dictionary Passing for Polytypic Polymorphism, by Juan Chen and Andrew W. Appel. Princeton University Computer Science TR-635-01, March 2001.

[11] Principled Scavenging, by Stefan Monnier, Bratin Saha, and Zhong Shao. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01), Snowbird, UT, pages 81-91, June 2001.

[12] Models for Security Policies in Proof-Carrying Code, by Andrew W. Appel and Edward W. Felten. Technical report, Princeton University Computer Science TR-636-01, March 2001.

[13] A Proof-Carrying Authorization System, by Lujo Bauer, Michael A. Schneider, and Edward W. Felten. Technical Report TR-638-01, Princeton University, April 2001.

[14] An Indexed Model of Recursive Types for Foundational Proof-Carrying Code, by Andrew W. Appel and David McAllester. ACM Transactions on Programming Languages and Systems 23 (5) 657-683, September 2001.

[15] Managing Memory with Types, Ph.D. Thesis, Daniel C. Wang, October 2001.

[16] A Type System for Certified Binaries, by Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 29th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'02), January 2002.

[17] A stratified semantics of general references embeddable in higher-order logic (extended abstract), by Amal Ahmed, Andrew W. Appel, and Roberto Virga. LICS'02: IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark, July 2002.

[18] A Syntactic Approach to Foundational Proof-Carrying Code, by Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, Zhaozhong Ni. LICS'02: IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark, July 2002.

[19] Type-Preserving Compilation of Featherweight Java, by Christopher League, Zhong Shao, and Valery Trifonov. ACM Transactions on Programming Languages and Systems 24(2) 112-152, March 2002.

[20] Foundational Proof-Carrying Code, by Andrew W. Appel. 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01), June 16-18, 2001.

[21] Functional Java Bytecode, by Christopher League, Valery Trifonov, and Zhong Shao. Workshop on Intermediate Representation Engineering for the Java Virtual Machine at the 5th World Multi-conference on Systemics, Cybernetics, and Informatics, July 2001.

[22] Mechanisms for Secure Modular Programming in Java, by Lujo Bauer, Andrew Appel, Edward Felten, in Software--Practice and Experience 33:461-480, 2003.

[23] Intensional Analysis of Quantified Types, by Bratin Saha, Valery Trifonov, and Zhong Shao. ACM Transaction on Programming Languages and Systems 25 (2) 159-209, March 2003.

[24] Precision in Practice: A Type Preserving Compiler for Java, by Christopher League, Zhong Shao, and Valery Trifonov. 12th International Conference on Compiler Construction (CC'03), April 2003. Lecture Notes in Computer Science Vol. 2622, Springer-Verlag.

[25] Supporting Binary Compatibility with Static Compilation. Dachuan Yu, Zhong Shao, and Valery Trifonov. In USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02), San Francisco, California, pages 165-180, August 2002.

[26] Inlining as Staged Computation, by Stefan Monnier and Zhong Shao. Journal of Functional Programming 13(3), pages 647-676, May 2003.

[27] Type-Preserving Compilation of Featherweight IL, by Dachuan Yu, Valery Trifonov, and Zhong Shao. In the International Workshop on Formal Techniques for Java-like Programs (FTfJP'02), Malaga, Spain, June 2002.

[28] A Type System for Certified Runtime Type Analysis, by Bratin Saha. Ph.D. Thesis, Department of Computer Science, Yale University, May 2002.

[29] A Type-Preserving Compiler Infrastructure, by Christopher Adam League. Ph.D. Thesis, June 2002, Department of Computer Science, Yale University, June 2002.

[30] A Trustworthy Proof Checker, by Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. Appeared (jointly) in Verification Workshop - VERIFY 2002 and in Foundations of Computer Security - FCS 2002, Copenhagen, Denmark, July 25-26, 2002.

[31] JVM TCB: Measurements of the Trusted Computing Base of Java Virtual Machines, by Andrew W. Appel and Daniel C. Wang. Princeton University CS TR-647-02, April 2002.

[32] More enforceable security policies, by Lujo Bauer, Jarred Ligatti, and David Walker. Foundations of Computer Security - FCS 2002, Copenhagen, Denmark, July 25-26, 2002.

[33] Policy-Enforced Linking of Untrusted Components (Extended Abstract), by Eunyoung Lee and Andrew W. Appel. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 371-374, September 2003.

[34] Typed Regions. Stefan Monnier and Zhong Shao. Technical Report YALEU/DCS/TR-1242, Department of Computer Science, Yale University, October 2002.

[35] Using Memory Errors to Attack a Virtual Machine, by Sudhakar Govindavajhala and Andrew W. Appel, 2003 IEEE Symposium on Security and Privacy, pp. 154-165, May 2003.

[36] Typed Machine Language, Ph.D. Thesis, Kedar N. Swadi, July 2003.

[37] Access Control for the Web via Proof-Carrying Authorization, Lujo Bauer, Ph.D. Thesis, August 2003.

[38] A Provably Sound TAL for Back-end Optimization, by Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. PLDI 2003: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 208-219, June 2003.

[39] Resource Usage Analysis Via Scoped Methods. Gang Tan, Xinming Ou and David Walker. Foundations of Object-Oriented Languages workshop, New Orleans, January 2003.

[40] The Logical Approach to Stack Typing. Amal Ahmed and David Walker. ACM SIGPLAN workshop on Types in Language Design and Implementation, New Orleans, January 2003.

[41] Building Certified Libraries for PCC: Dynamic Storage Allocation, by Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. 2003 European Symposium on Programming ESOP'03), Warsaw, Poland, April 2003.

[42] Secure Linking: A Logical Framework for Policy-Enforced Component Composition. Eun-Young Lee, Ph.D. Thesis, December 2003.

[43] A Syntactic Approach to Foundational Proof-Carrying Code (Extended Version), by Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. Journal of Automated Reasoning 31 (3-4): 191-229, 2003.

[44] Intensional Analysis of Higher-Kinded Recursive Types, by Gregory Collins and Zhong Shao. Technical Report YALEU/DCS/TR-1240, Department of Computer Science, Yale University, October 2002.

[45] Dependent Types Ensure Partial Correctness of Theorem Provers, by Andrew W. Appel and Amy Felty. J. Functional Programming 14(1):3-19, January 2004.

[46] A Type System for Certified Binaries. Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. In Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02), pages 217-232, January 2002.

[47] An Encoding of Fomega in LF, by Carsten Schuermann, Zhaozhong Ni, and Hai Fang. 2001 Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN'01), Siena, Italy, June 2001.

[48] Reasoning about Hierarchical Storage. Amal Ahmed, Limin Jia and David Walker. IEEE Symposium on Logic in Computer Science, pp. 33-44, June 2003.

[49] Edit Automata: Enforcement Mechanisms for Run-time Security Policies. Jay Ligatti, Lujo Bauer and David Walker. Accepted for publication (June 2004), International Journal of Information Security.

[50] A Meta Linear Logical Framework, by Andrew McCreight and Carsten Schuermann. Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04) July 5, 2004.

[51] Building Certified Libraries for PCC: Dynamic Storage Allocation (Extended Version), by Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Invited submission to Science of Computer Programming, Elsevier, May 2003.

[52] Simulating Quantified Class Constraints, by Valery Trifonov. 2003 ACM Haskell Workshop, August 2003.

[53] Principled Compilation and Scavenging, by Stefan Monnier. Ph.D. Thesis, June 2003, Department of Computer Science, Yale University.

[54] Foundational Proof Checkers with Small Witnesses, by Dinghao Wu, Andrew W. Appel, and Aaron Stump. 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, August 2003.

[55] A Low-Level Typed Assembly Language with a Machine-checkable Soundness Proof, Juan Chen, Ph.D. Thesis, January 2004.

[56] Semantics of Types for Mutable State. Amal J. Ahmed, Ph.D. Thesis, July 2004.

[57] Network Security Management with High-level Security Policies, by Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel, Technical report TR-714-04, Princeton University, 2004.

[58] Construction of a Semantic Model for a Typed Assembly Language, by Gang Tan, Andrew W. Appel, Kedar N. Swadi, and Dinghao Wu. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04), January 2004.

[59] Dynamic Typing with Dependent Types (extended abstract). Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 3rd IFIP International Conference on Theoretical Computer Science, August 2004.

[60] A general and flexible access-control system for the web. Lujo Bauer, Michael A. Schneider, and Edward W. Felten. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[61] Policy-based Multihost Multistage Vulnerability Analysis, by Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Submitted for publication, November 2004.

[Ellison 1999] *SPKI Certificate Theory,* by C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. RFC2693, September 1999.

[Housley 2002] *Internet X.509 Public Key Infrastructure Certificate and CRL Profile,* by R. Housley, W. Polk, W. Ford, and D. Solo. RFC3280, April 2002.

[Joseph 2003] Control Techniques to Eliminate Voltage Emergencies in High Performance Processors, by Russ Joseph, David Brooks, and Margaret Martonosi. *The Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

[Lampson 1992] Authentication in distributed systems: Theory and practice, by B. Lampson, M. Abadi, M. Burrows, and E. Wobber. ACM Transactions on Computer Systems 10(4)265-310, November 1992.

[MacKenzie 2001] *Mechanizing Proof,* by Donald MacKenzie. MIT Press, 2001.

[Microsoft 2002] Microsoft Security Bulletin MS02-65, Microsoft Corporation, http://www.microsoft.com/technet/security/bulletin/MS02-065.asp, November 2002.

[Wojcik 2003] Introduction to OVAL: A new language to determine the presence of software vulnerabilities, by Matthew Wojcik, Tiffany Bergeron, Todd Wittbold, and Robert Roberge. http://oval.mitre.org/documents/docs-03/intro/intro.html, November 2003.